# POLYCHRONY

## A TOOLSET FOR SIGNAL (SSME Platform)

# Polychrony SSME User Guide

V0

| | Author(s) | Checked by | Approbation |
|---|---|---|---|
| **Name** | **Members of the Espresso Team** | **Loïc Besnard****<br>**Thierry Gautier**<br>**Paul Le Guernic** | **Jean-Pierre Talpin** |
| **Company** | **INRIA** | **INRIA, **CNRS** | **INRIA** |
| **Department** | **Espresso Team** | **Espresso Team** | **Espresso Team** |
| **Date** | | | |
| **Visa** | | | |
| **Summary** | **User Guide of the Polychrony SSME, a front-end to the Signal ToolBox in the Eclipse environment.** | | |

**Attention** : la responsabilité des entreprises et des organismes ayant participé à l'élaboration de ce document ne peut en aucun cas être engagée en cas de dommages ou de pertes résultant de l'utilisation ou de l'exploitation des informations qui y sont contenues.

**Disclaimer** : Contractors participating to this report shall incur no liability whatsoever for any damage or loss which may result from the use or exploitation of information and/or Rights contained in this report.

# Table of Contents

1 Preface....3
1.1 Table of versions....3
1.2 Table of references and applicable documents....3
1.3 Acronyms and glossary....3
2 Subject....4
2.1 Purpose of the document....4
2.2 Editing particularities....4
2.2.1 Changes identification....4
2.2.2 Temporary editing....4
2.3 Application scope....4
2.4 Edition and evolution of the document....4
3 Context....4
4 Using the SSME Platform....6
4.1 Creation of a new project....7
4.2 Creation of a new SSME Model....7
4.3 The compilation scenarios....8
4.3.1 Functionalities....8
4.3.2 Generators....9
4.3.3 Creation of a compilation scenario....11
4.3.4 SSME Scenario View....11
4.4 The Signal text editor....11
4.5 Connection to Polychrony services....12
4.5.1 How does the connection work ?....12
4.5.2 Compiler configuration....13
4.5.3 Applying Polychrony services....15
4.5.4 Simulation....18
4.5.4.1 C/C++....18
4.5.4.2 Java....18

# Index of Illustrations

# 1 Preface

## *1.1 Table of versions*

| *Version* | *Date* | *Description & rationale of modifications* | *Sections modified* |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

## *1.2 Table of references and applicable documents*

| *Reference* | *Title & edition* | *Author or editor* | *Year* |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

## *1.3 Acronyms and glossary*

| *Term* | *Description* |
|---|---|
| SSME | Signal Syntax Model under Eclipse. |

# 2 Subject

## *2.1 Purpose of the document*

This document is a user manual for the SSME Platform, a component of the Polychrony Toolset. It is a front-end to the Signal ToolBox in the Eclipse environment.

## *2.2 Editing particularities*

### 2.2.1 Changes identification

All the changes made since the previous publication are identified using the sign | in the left margin of each line holding a modification.

### 2.2.2 Temporary editing

Special points are signaled like this :

. ***temporary***

. ***incomplete***

. ***to be defined***

. ***to be confirmed***

## *2.3 Application scope*

## *2.4 Edition and evolution of the document*

Tell here who is responsible of the evolution of the document and when it is necessary to do it.

This documentation must be updated when the meta model (SSME) is modified and when a functionality of the Signal ToolBox is added to the SSME Platform.

# 3 Context

The Polychrony toolset, based on *Signal*, is a development environment for critical systems, from abstract specification until deployment on distributed systems. It relies on the application of formal methods, allowed by the representation of a system, at the different steps of its development, in the Signal polychronous semantic model. It provides a formal framework:

- to validate a design at different levels,
- to refine descriptions in a top-down approach,
- to abstract properties needed for black-box composition,
- to assemble predefined components (bottom-up with COTS).

*Signal* is based on synchronized data-flow (flows + synchronization): a process is a set of equations on elementary flows describing both data and control, the variables of the

system are signals. A signal is a sequence of values which has a clock associated with; this clock specifies the instants at which the values are available.

The *Signal* formal model provides the capability to describe systems with several clocks (polychronous systems) as relational specifications. Relations are useful as partial specifications and as specifications of non-deterministic devices (for instance a non-deterministic bus) or external processes (for instance an unsafe car driver).

Using *Signal* allows to specify an application, to design an architecture, to refine detailed components down to RTOS or hardware description. The *Signal* model supports a design methodology which goes from specification to implementation, from abstraction to concretization, from synchrony to asynchrony.

The principal application areas for the *Signal* language are that of embedded, real-time, critical systems. Typical domains include:

- Process control,
- Signal processing systems,
- Avionics,
- Automotive control,
- Vehicle control systems,
- Nuclear power control systems,
- Defense systems,
- Radar systems...

It constitutes a development environment for critical systems, from abstract specification until deployment on distributed systems. It relies on the application of formal methods, allowed by the representation of a system, at the different steps of its development, in the Signal polychronous semantic model.

For more information concerning the INRIA Polychrony environment, consult the ESPRESSO team website: http://www.irisa.fr/espresso/Polychrony

The POLYCHRONY TOOLSET (See Illustration 1) contains three main components:

- The Signal ToolBox, a batch compiler for the SIGNAL language, and a structured API that provides a set of program transformations. The Signal ToolBox can be installed without the other components.
- The Signal GUI, a Graphical User Interface to the Signal ToolBox (editor + interactive access to compiling functionalities). Signal GUI requires the Signal ToolBox ( or an other component that redefines the Signal ToolBox ApIs).
- The SSME PLATFORM, a front-end to the Signal ToolBox in the ECLIPSE environment. SSME PLATFORM requires the Signal ToolBox (or an other component that redefines the Signal ToolBox ApIs). SSME stands for Signal Syntax Model under Eclipse.

The POLYCHRONY TOOLSET, also provides:

- libraries of SIGNAL programs,
- a set of SIGNAL programs examples,
- user oriented and implementation documentations,
- facilities to generate new versions.

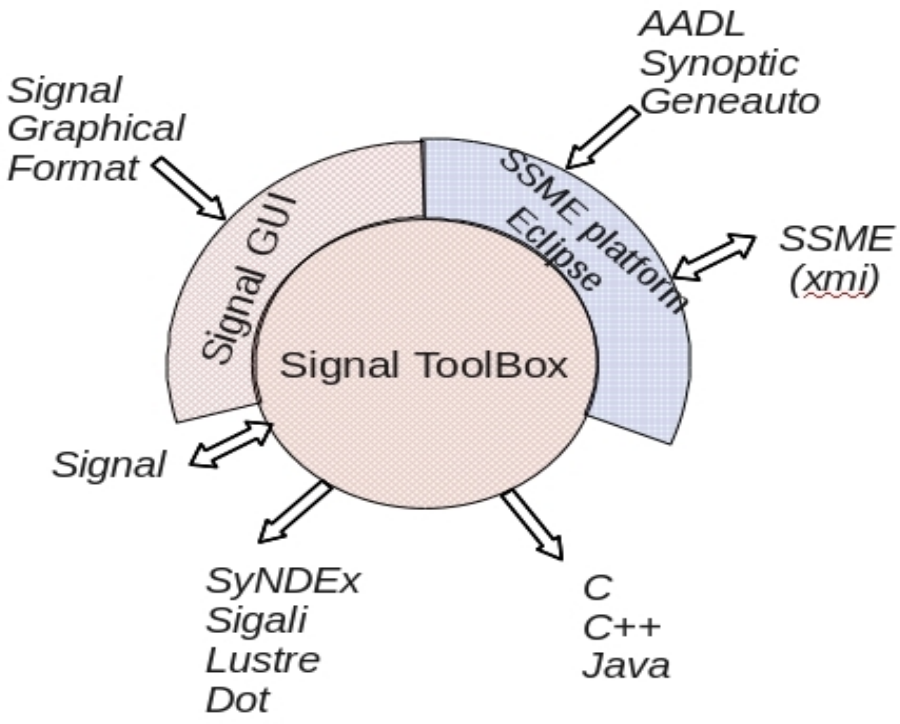



*Illustration 1: Polychrony tool set*

# 4 Using the SSME Platform

The SSME user guide is composed of the following sections:

- Creation of a new project
- Creation of a new SSME Model
- The compilation scenarios
- The Signal text editor
- Connection to Polychrony services

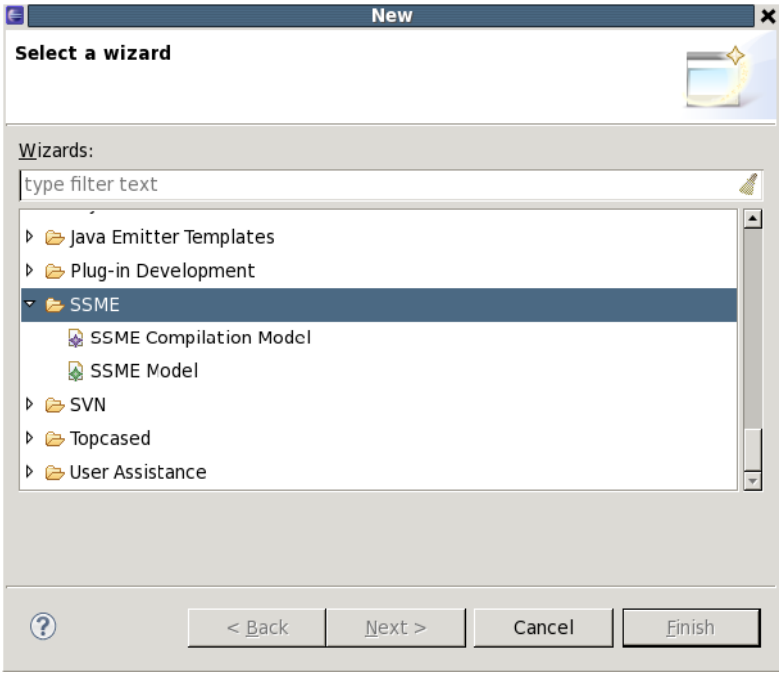
## *4.1 Creation of a new project*

If you have no project in your workspace, you will need to create a new one:

- **Right-click** on the navigator view
- select **New -> Project ...**
- Select then a **General project**, this will be sufficient to model, and click on **Next >** :
- Give a name to the project and click on **Finish**

## *4.2 Creation of a new SSME Model*

To help the user during the creation of a new SSME model file, the reflexive editor has a wizard. To start the wizard, right-click on the project where you want to create your model, and select **New-> Other...** and then select the following model file : **SSME -> SSME Model.**




Once you have selected the **SSME Model**, the SSME wizard is opened. Then, you have to choose the name of the diagram. Select **Next>** and then select the kind of the root for the model file. There are three kinds of root model elements :

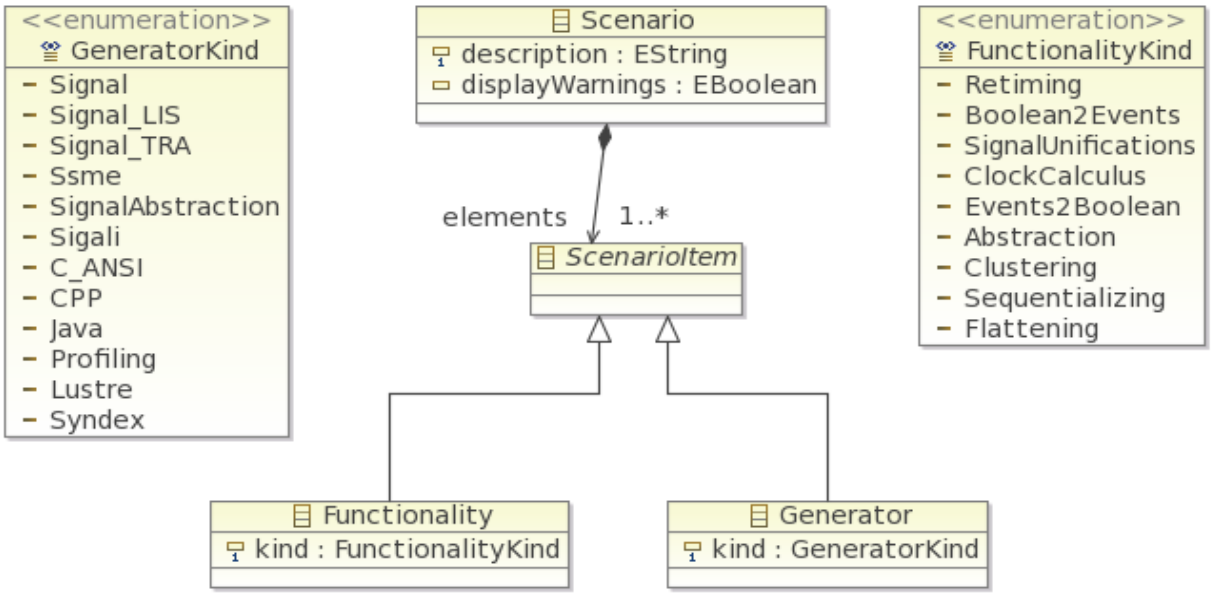
- List Module : it corresponds to a library of components, types, and constants.
- Process Model : it corresponds to a SIGNAL component (process, node, action, function).
- Operator Model : it corresponds to a SIGNAL operator definition.

Once you have chosen the root model, you only have to add new model objects. To do so, right-click on the node on which you want to add a child, and select the **New Child**

menu. It displays the list of all possible model elements that can be added for the current selected element. If this option does not exist for a node, it means that there is no possible child for this node.

## *4.3 The compilation scenarios*

This plug-in allows the creation of compilation scenarios for a Polychrony model. A compilation scenario is composed of different kinds of functionalities and generators. A functionality modifies the internal representation of the program; a generator translates this representation into a specific external format. The meta-model of this plug-in is shown on the next picture.




### 4.3.1 Functionalities

% This part must be generated from the Signal ToolBox functionalities %

*** to be updated***

The functionalities that can be applied are those of the Polychrony Signal ToolBox. For the moment, the following functionalities can be applied to a program:

- **Retiming**: It performs a shift register transformation. It rewrites synchronous function f such that Y := f(X1 $ m1 init V1,...,Xn $ mn init Vm) into Y := y' $ j init f(V1',...,Vm') and y' := f(X1 $ m1' init V1",...., Xm $ mn' init Vm").
- **Booleans to events**: It rewrites boolean expressions of under-sampling for logical and relational operators. the goal is to prove synchronization constraints of the system.
  - the rewriting of the booleans expressions referenced in an extraction (*when E*) when E is not a "free" condition and not declared as an assumption. For example, the expression *when (A and B)* is equal to the clock intersection of

*when (A)* and *when (B)*.
So, "classical boolean rules" completed with some specific rules induced by the extraction and merging Signal operators are applied to the system.

- the rewriting of the boolean expressions referenced in the extraction (when E) when E is assumed to be an assertion: the expression *assert(E)* specifies that E is always true when it is available. So it induces that when E is equivalent to the clock of E.

- **Signal unifications**: this operation consists in the merge of nodes into one node when their definition expression are equal (syntactically).
- **Clock calculus**: it performs the resolution of the clock systems using a triangularization technique. A BDD-based data structure is used. Here, only the study of static properties is performed. They allow to characterized the set of states in which the automaton associated to a program can evolve, independently of initial values, and the set of the transitions between these states. The result is a forest of clock trees.
- **Events to booleans**: it performs the inter-format DC+ (that constrains event objects, forest of trees of clocks) to bDC+ (without event objects, a tree of boolean clock) translation.
- **Abstraction**: it computes the abstraction of the program (I/O data dependences, I/O clock relations, the "black Box" or the "grey Box" abstraction representation). This abstraction is useful for separated compiling.
- **Sequential clustering**: it performs the following partitioning (called input train). Two nodes are in the same set if and only if they depends on the same subset of inputs signals of the graph. The graph is modified. The nodes are clustered into sub-graphs. The internal representation must be in bDC+ sub-format.
- **Sequentializing** : it performs the inter-format bDC+ (a tree of boolean clocks) to sbDC+ («sequentialized boolean *dc+*») translation. The internal representation must be in bDC+ sub-format. The nodes of the internal representation are ordered. The assert nodes are visible for code generation (code will be generated for verifying assumptions at run time).
- **Flattening** (STS): it performs the inter-format bDC+ (a tree of boolean clocks) to STS (a tree of boolean clocks reduces to one level) translation. The internal representation must be in bDC+ sub-format.

### 4.3.2 Generators

% This part must be generated from the Signal ToolBox functionalities %

*** to be updated***

There are 12 kinds of generators:

- **Signal Textual** (SIG): it translates the internal representation of the specification into a textual Signal file (.sig file).
- **Signal Textual** (LIS): it translates the internal representation of the specification

into a textual Signal file (.sig file), after the graph creation. The errors will be written on the file.

- **Signal Textual** (TRA): it translates the internal representation of the specification into a textual Signal file (.sig file), after the clock calculus phase.
- **Signal Model** (Ssme): it translates the internal representation of the specification into a XMI Model Signal file (.ssme file). This model file is conformed to the SSME meta-model.
- **Signal Abstraction**: it translates the abstraction of the internal representation into a textual Signal file (.sig file).
- **Sigali**: it translates the internal representation of the specification into a textual Sigali file (.z3z file). This file is then used by Sigali to prove dynamical properties.
- **C ANSI**: it translates the internal representation of the specification into a textual C ANSI files (.c and .h files). These files are used to simulate the Signal specification. It is applied to a graph which must be a sbDC+ one. The graph is with or without clusters.
- **C++**: it translates the internal representation of the specification into a textual C++ files (.cpp and .h files). These files are used to simulate the Signal specification. It is applied to a graph which must be a sbDC+ one. The graph is with or without clusters.
- **Java**: it translates the internal representation of the specification into a textual Java files (.java files). These files are used to simulate the Signal specification. It is applied to a graph which must be a sbDC+ one. The graph is with or without clusters.
- **Profiling**: it produces the morphism of the internal representation according to the definitions assigned of the Signal operators given in the "ht" table into a textual Signal file (.sig file).
- **Lustre**: it translates the internal representation of the specification into a textual Lustre file (.lus file).
- **Syndex**: it translates the internal representation of the specification into a textual SynDEx file (.sdx file) for code distribution.

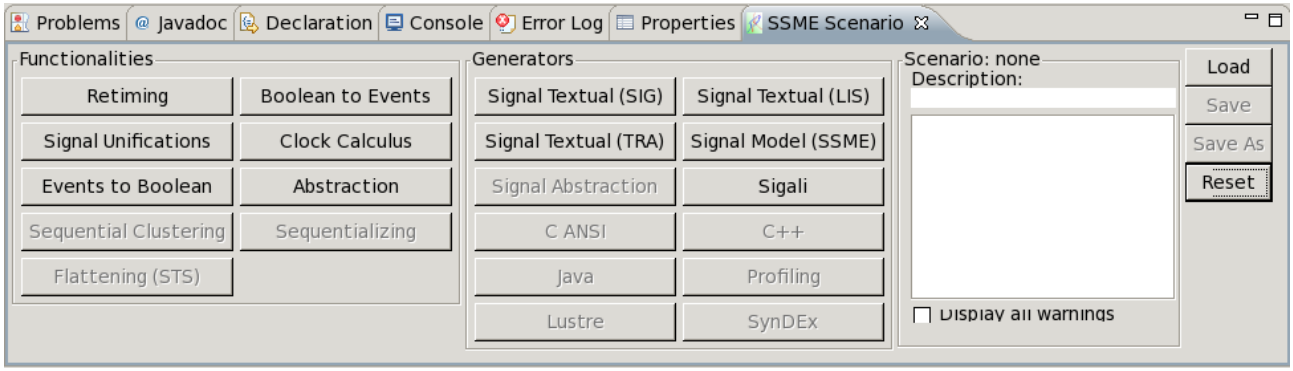
### 4.3.3 Creation of a compilation scenario

The reflexive editor has been automatically generated from the compilation scenario meta-model. To create a new compilation scenario file (.ssc), right-click on your project and select **New->Other...** and then select the following model : **Polychrony->SSME Compilation Model.**

However, there are some constraints to create a compilation scenario, because some functionalities/generators can only be applied after others, so an interactive view (described in next part) has been created to help user to create such scenario.

### 4.3.4 SSME Scenario View

The SSME Scenario View (see following picture) constitutes a way to describe a compilation scenario with some assistance. Each functionality and generator is represented by a button and, according to the functionality or generator you activate, others become available or are disabled.

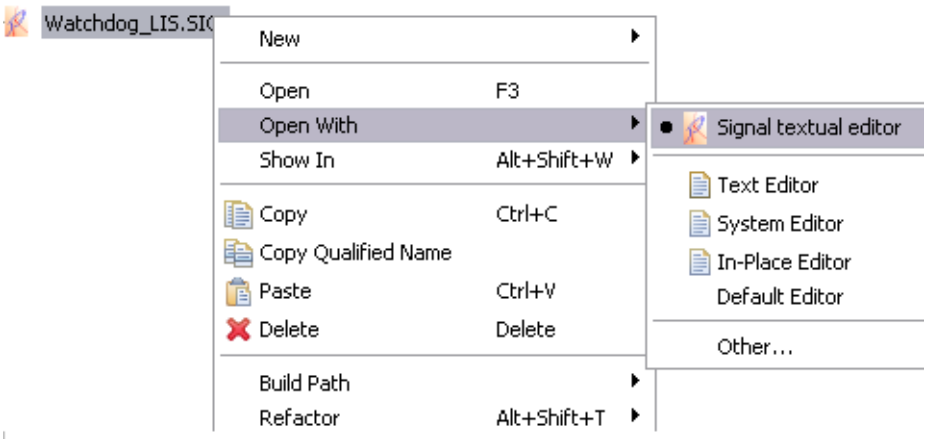



To access to this view, select **Window-> Show View-> Other...**, and then select **Polychrony->SSME Scenario**.

## *4.4 The Signal text editor*

The SSME environment integrates a simple text editor to manipulate Signal Text files under Eclipse. The current version provides only syntax highlighting for Signal keywords, for comments, and for constant value using primitive types (string, character, or numerical value).

To use this editor, you have only to double-click on the file with extension .SIG. Maybe the Signal Text editor is not selected by default, so, you have to right-click on the file and select **Open With->Signal Text Editor**.



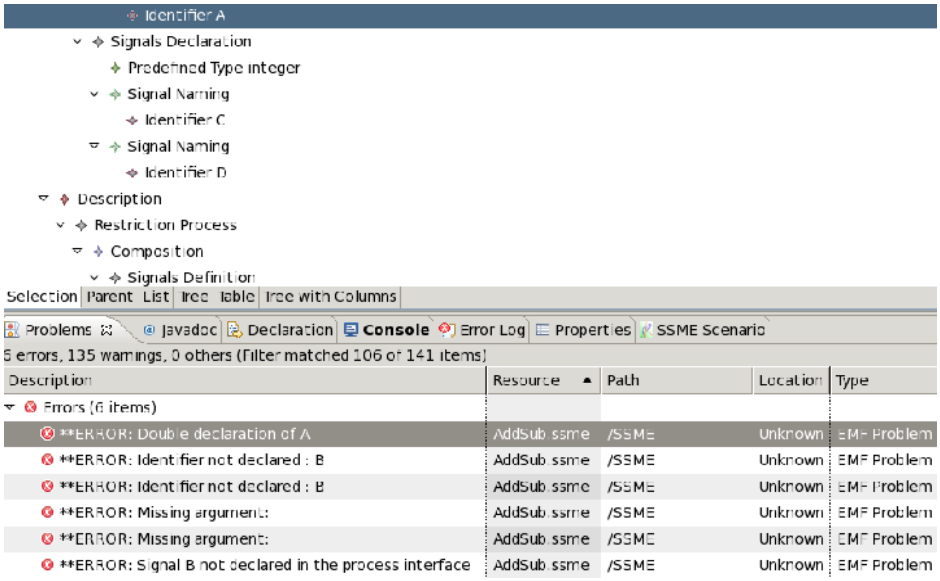

## *4.5 Connection to Polychrony services*

To access the Polychrony services under Eclipse, the compiler has been deeply connected to the reflexive editor and the graphical environment. We obtain a traceability between the SSME models and the results returned by the compiler. It becomes possible to show the compilation errors directly on the source model.

### 4.5.1 How does the connection work ?

The connection between the SSME Eclipse editors and the POLYCHRONY compiler is a Java/C JNI interface to communicate with the compiler through native libraries. The principle of the communication is the following:

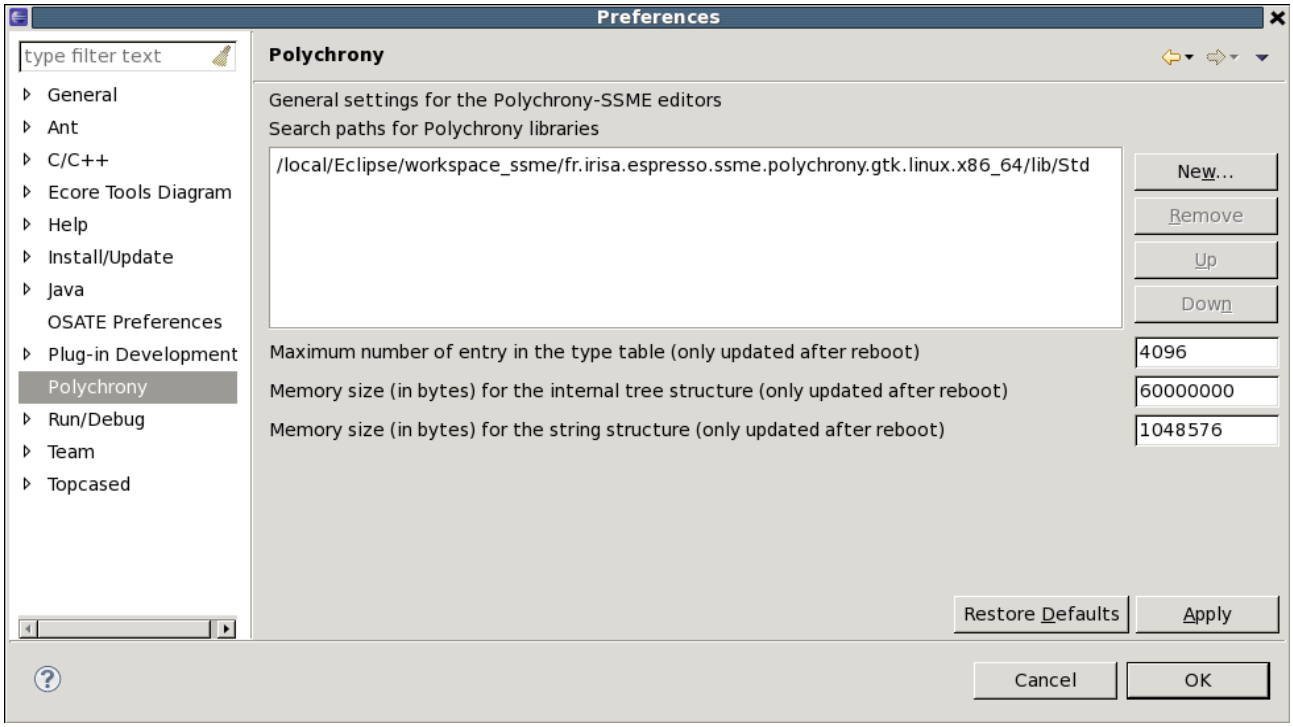
1. First, the SSME model is transformed into the abstract syntax tree (AST) representation inside the compiler. A SSME model parser that makes this translation has been developed.
2. Then, this AST representation is transformed into an internal graph structure. This step consists in resolving all references specified inside the AST, checking the type errors, and making explicit all implicit clock constraints and clock relations. This means that new signals, which do not exist in the source model, are created. The traceability consists in linking each new signal to the corresponding original AST object.
3. The third step consists in applying to this graph the different POLYCHRONY services specified in the compilation scenario (clock resolution, code generation...). These operations also modify the graph obtained at the previous step.
4. Finally, by analyzing the graph obtained after these transformations, some errors can be reported in the model.


Identifier A
Signals Declaration
Predefined Type integer
Signal Naming
Identifier C
Signal Naming
Identifier D
Description
Restriction Process
Composition
Signals Definition
Selection Parent List Tree Table Tree with Columns
Problems Javadoc Declaration Console Error Log Properties SSME Scenario
6 errors, 135 warnings, 0 others (Filter matched 106 of 141 items)
Description Resource Path Location Type
Errors (6 items)
**ERROR: Double declaration of A AddSub.ssme /SSME Unknown EMF Problem
**ERROR: Identifier not declared : B AddSub.ssme /SSME Unknown EMF Problem
**ERROR: Identifier not declared : B AddSub.ssme /SSME Unknown EMF Problem
**ERROR: Missing argument: AddSub.ssme /SSME Unknown EMF Problem
**ERROR: Missing argument: AddSub.ssme /SSME Unknown EMF Problem
**ERROR: Signal B not declared in the process interface AddSub.ssme /SSME Unknown EMF Problem

A Hashmap between the Polychrony native AST and the SSME instances has been built, in order to associate the node of the native AST where the error message is written and the corresponding instance. For example, here, the “Double declaration of A” error is linked with the highlighted Identifier A.

## 4.5.2 Compiler configuration

The Signal compiler can be parameterized to specify the memory size used for its operations. To access to these parameters, select the Polychrony section in the Preferences Window as shown in the following picture.



There are four parameters:

- **The Polychrony library path.** Here, you add all paths in which there are .LIB or .SIG file that contain the Signal module needed for the compilation. A lib directory that contains all libraries provided with the classical Polychrony environment is available in a plug-in directory named:
  - *fr.irisa.espresso.ssme.polychrony.win32.win32.x86* for Windows x86
  - *fr.irisa.espresso.ssme.polychrony.win32.win32.x86_64* for Windows x86_64
  - *fr.irisa.espresso.ssme.polychrony.gtk.linux.x86* for Linux x86
  - *fr.irisa.espresso.ssme.polychrony.gtk.linux.x86_64* for Linux x86_64
  - *fr.irisa.espresso.ssme.polychrony.cocoa.macosx.x86* for MacOS X/Intel x86
  - *fr.irisa.espresso.ssme.polychrony.cocoa.macosx.x86_64* for MacOS X/Intel x86_64

  By default, this parameter only contains the root of this lib directory. For example, if you want to use the apex-arinc library, add to the parameter the path *<lib-directory>/apex_arinc653/lib*.
- **Maximum number of types.** This parameter represents the maximum number of types that can be loaded during a compilation. Default value is 4096.
- **Memory size for the internal structure.** This parameter represents the memory size reserved for the allocation of the internal tree structure used during a compilation. Default value is 60000000 bytes.
- **Memory size for the string structure.** This parameter represents the memory size reserved for the allocation of strings. Default value is 1 Mb.

If you change one of these values, validate these changes by clicking on the **apply** button. For restoring the initial values, click on the **Restore Defaults** button.

### 4.5.3 Applying Polychrony services

To call Polychrony services, right-click on the SSME file on which you want to apply the service(s), and select **Polychrony.**
You can choose to use a predefined scenario : in this case, select the **Execute Predefined Scenarios** option.

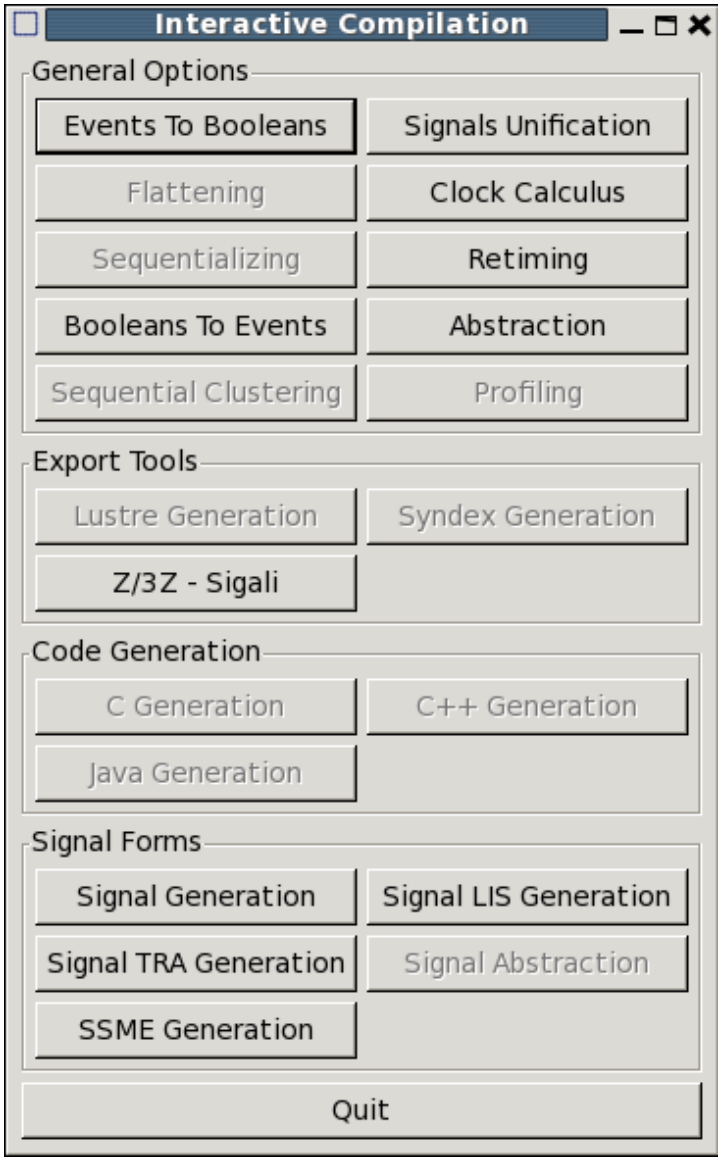
Seven actions are proposed:

- Generate C files from the selected SSME model
- Generate C++ files from the selected SSME model
- Generate Java files from the selected SSME model
- Generate SIG file from the selected SSME model
- Generate SIG (LIS) file from the selected SSME model
- Generate SIG (TRA) file from the selected SSME model
- Execute a compilation scenario on the model

The three first actions generate the executable code files respectively in C, C++, Java language from the SSME model specifications. The next three actions generate a Signal textual file : a normal SIG file, a pretty printing of the tree internal structure or a Signal file generated during the transformation to the internal graph structure. The last option applies a compilation scenario on the SSME model (see previous part). All these files are generated only in a directory named as the SSME model file. They are only generated if all previous compilation steps are all successful.

It is also possible to call Polychrony services on SIG file. Right click on the SIG file, and select **Polychrony** to obtain all actions proposed for the SSME file and one action to generate a SSME file from the SIG file in a directory named as the SIG file.

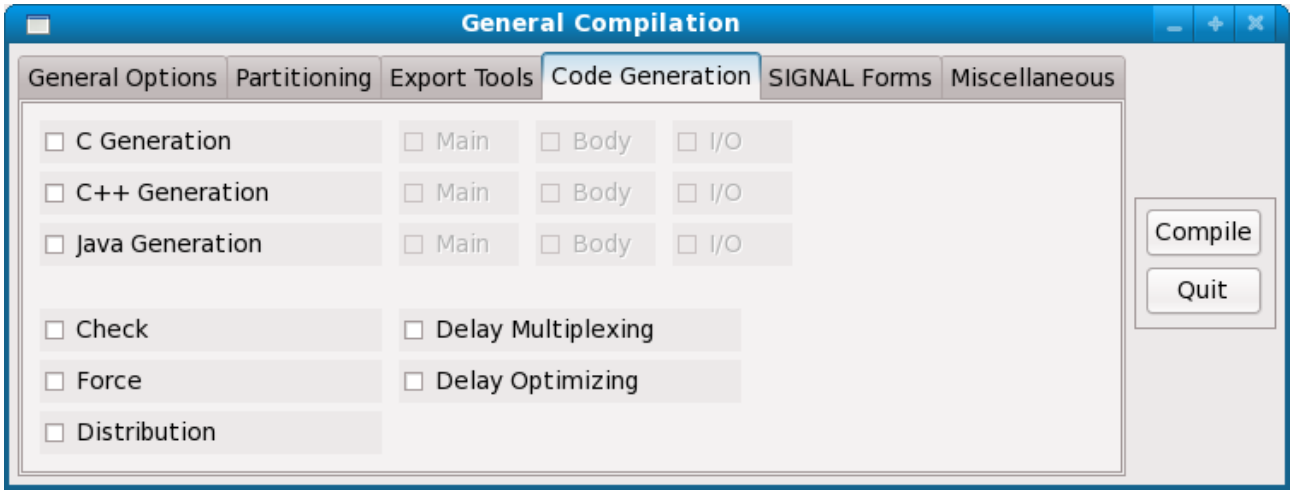
There are two more options : the interactive compilation and the general compilation. Let's see the interactive compilation first : to be able to use it, choose the **General Compilation** menu in the **Polychrony** menu, and then click on **Interactive Compilation**. A new window will pop up, with all the options available in the scenario editor.




Each time you click on an option, the corresponding operation is realized in real-time. When you have finished, you can close this window by clicking on the **Quit** button.

To use the general compilation GUI, choose the **General Compilation** menu in the **Polychrony** menu, and then click on **General Compilation**.

A GUI with tabs will pop up :




Each time you're clicking or unclicking an option, the option manager of the SIGNAL compiler is called and the corresponding options are activated or unactivated. The compilation starts when you push the “Compile” button.

### 4.5.4 Simulation

The Polychrony services provides several kinds of generators which needs external tools to be used or executed: SynDEx tools for sdx files, Sigali (distributed with the classical Polychrony distribution) for z3z files... The C, C++ or Java code generation is dedicated to simulations.

By default, the code (C, C++, or Java) generated by Polychrony will read input values from a files called R<input name>.dat (or RC_<input name>.dat if the input signal is of type event) and will write output values into a file called W<output name>.dat. These input files has to be created and filled by the user (one value per line, and for event signal, 0 for absence and 1 for presence).

## *4.5.4.1 C/C++*

For C and C++, you will need to use an external C/C++ compiler. The Polychrony libraries are provided and can be found in the Eclipse **plugins** directory and specifically in:

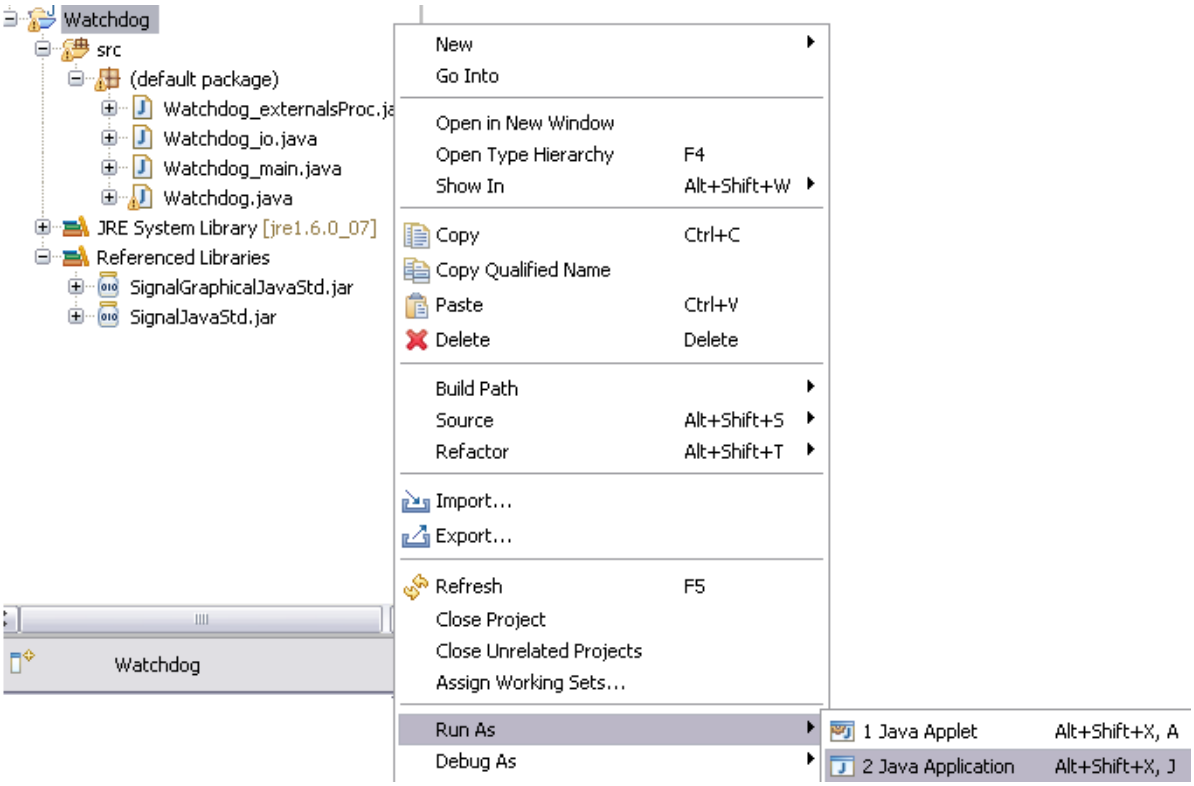
- *fr.irisa.espresso.ssme.polychrony.win32.win32.x86/lib* for Windows x86
- *fr.irisa.espresso.ssme.polychrony.win32.win32.x86_64/lib* for Windows x86_64
- *fr.irisa.espresso.ssme.polychrony.gtk.linux.x86/lib* for Linux x86
- *fr.irisa.espresso.ssme.polychrony.gtk.linux.x86_64/lib* for Linux x86_64
- *fr.irisa.espresso.ssme.polychrony.cocoa.macosx.x86/lib* for MacOS X/Intel x86
- *fr.irisa.espresso.ssme.polychrony.cocoa.macosx.x86_64/lib* for MacOS X/Intel x86_64

Currently, there is not automatic generation of makefile to compile C/C++ file.

## *4.5.4.2 Java*

For the Java generation code, we take benefit of Eclipse which is originally a Java environment which provides several facilities to manage Java project. The call of the Java generation code creates (if all previous operations are executed without errors) a Java project containing the Polychrony Java libraries (**SignalJavaStd.jar**, and **SignalGraphicalJavaStd.jar**) and the Java source classes are generated in the **src** directory of the project. There may be some errors in the Java code if your SSME model has constant parameters. The correction of these errors consists in replacing the **UNDEF()** calls by the value of the constant parameter.

To execute the simulation, you have only to right-click on the project (or on the Java class corresponding to the main) and select **Run As->Java Application**.


Watchdog
src
(default package)
Watchdog_externalsProc.ja
Watchdog_io.java
Watchdog_main.java
Watchdog.java
JRE System Library [jre1.6.0_07]
Referenced Libraries
SignalGraphicalJavaStd.jar
SignalJavaStd.jar
Watchdog
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
1 Java Applet Alt+Shift+X, A
2 Java Application Alt+Shift+X, J